Fernando Faria

10/31/15

Dr. Knox

**Virtual Memory Project Lab Report**

***Abstract:***

Virtual Memory is a prominent technology in operating systems today. A virtual address space is seen by the program as the memory is resides in. The operating system and hardware see the physical address space. It is up to the Operating System to manage the translation from virtual to physical.

Managing memory usage and scheduling processes for execution time are the main duties of an Operating System. This lab was broken into four parts.

In part 1, I created a program to generate reference strings that represent the memory requests a program would make in its own virtual space. My generation program features four types of locality: spatial, temporal, spatial-temporal, and random.

In part 2, I discuss which replacement algorithm should be best suited for each locality. I considered FIFO, LRU, and Second Chance.

In part 3, I created a program to simulate the effects of those three replacement algorithms on the generated reference strings from part 1. I generated statistics to show the performance of those replacement algorithms on those reference strings.

In part 4, I expanded upon my simulation to provide a multiprogramming environment with preemptive round robin scheduling. I only implemented one thread of execution.

***Problem Statement:***

Determining the most efficient demand paging replacement algorithm is not an easy task. Sometimes it is necessary, and only possible, to solve a problem by actually implementing it and analyzing the results. This is the purpose of this lab.

***Part 1:***

This demonstrates the spatial locality generated by my reference string generator. The size of the string generated is 250 characters. As you can see, the x axis goes up to 250. The range of pages generated by my generator is from one to 100. In order to visualize the locality, you must think of the x axis as time. From time 1-8, you can see that pages at 60 +/- 5 were being generated. After a certain amount of time, the locality jumps to around 80 +/- 5. After that, the locality goes back to the previous locality centered at 60. Zigzagging lines actually represent the random page requests in one locality, while a sharp vertical line shows a change in locality.

My algorithm starts by randomly generating a locality size. After it chooses one, it chooses a center for that locality, adjusting for the program’s bounds. It also randomly generates a length for that locality, and it keeps generating page requests only inside that locality until it reaches that length. It then has a 50% chance to jump back to the previous locality, or generate a new one.

The temporal locality is apparent in this graph. As with the previous graph, time is on the x axis and the variation in the y axis represents a new page request. Once a page has been requested, there is no higher probability that a page nearby will also be requested. However, there are many straight lines in this graph, which is the temporal locality at work. Once a page has been generated, that page has a higher chance to be requested again before another page is requested.

My algorithm is as follows. I create a sorted singly linked list of pages, with the head of the list at the page with the largest “visit” count. Initially, no page has a larger “visit” count, but all pages are initialized to INT\_MAX so that any initially requested page will be smaller than a page not requested yet. I then perform the modulus operation between a random number much larger than the size of the reference string, by twice the size of the reference string. I then traverse the linked list until I visit the amount of nodes I just generated, or I reach the end of the list. While I do this traversal, I increment the “visit” count of each node I pass. The last node I visit is the node I request, as it will either be at the end of the list, or at a position before the end of the list that coincides with the number I randomly generated with the modulus operation. I then place this requested node at the tail of the list, because it’s “visit” count is the smallest in the entire list. Since I chose the operand for the modulus operation to be twice the size of the reference string, there is always a 50% chance I will request the previously requested page again.

The downside of this is that the 2nd most recently requested page has an equal chance to be requested as any other page. However, once it is chosen, it then has a large chance again to be requested anew.

This graph shows a merge of the two prominent patterns visible in the previous two graphs. The small, straight horizontal lines represent temporal locality, as that page is being requested multiple times in a row. The zigzagging pattern shows the spatial locality, as pages near a recently requested page are more likely to be requested than a page outside of that locality.

My algorithm is also a merge of the two algorithms above. First, I choose a spatial locality using the exact same algorithm as the one used above to generate only spatial locality strings. However, once this locality is defined, I hand it off to the temporal locality function to create the small strings of repeating page requests.

There is no locality in this graph. Each page request is randomly generated and there is nothing that can be analyzed from past requests to allow us to make a better guess at which pages are safer to keep.

The distance between one request to the next is random, thus proving that this was generated randomly.

***Part 2***

Optimal Page Replacement Strategy

1. Spatial Locality

For a program that exhibits strong spatial locality, LRU performs better than FIFO. Considering that the locality of a program sometimes jumps back to a previous locality, FIFO would not be able to prevent those early-requested page from being paged out when locality returns to them.

LRU also performs well compared to Second Chance for a program that exhibits strong spatial locality. Second Chance also assumes that a previous locality, if not returned to soon enough, is as safe to be paged out as a locality that could have been in focus much less recently. LRU is able to distinguish between localities that were very recently used and not very recently used, which helps postpone costly page faults if a recent previous locality is returned to.

1. Temporal Locality

Programs that exhibit strong temporal locality perform better in systems that utilize FIFO. This is under the assumption that the program does not strongly exhibit other forms of locality. Temporal locality is inherently less likely to be returned to than spatial locality. If a program starts requiring other pages to be swapped in, this need is proportional to the decreasing need of the pages currently in the system, which is known as a change in locality. As more time accumulates between when a locality was last used and the present, it is more likely that a temporal locality will not be needed again.

Compared to LRU, FIFO is much less overhead. It only requires a pointer to a list that does not need to be moved. LRU would require keeping track of new and old pages, and the times they were last used, and performing calculations on those times in order to find the least recently used page.

FIFO performs better than Second Chance for a program that predominantly exhibits temporal locality, also due to the overhead Second Chance costs when it periodically grants all processes a second chance to stay in memory. This overhead is costly.

1. Spatial and Temporal Locality

A program that exhibits strong spatial and temporal locality performs best with the Second Chance algorithm. The Second Chance algorithm, along with the LRU algorithm, both attempt to determine which pages are safest to page out. A program exhibiting spatial locality does not do well with a FIFO algorithm, due to the common occurrence that previous localities are revisited, and FIFO would consider these safe to page out.

The LRU costs more to operate. The Second Chance algorithm only requires 1 bit to keep track of safe vs unsafe, while the LRU ranks pages, and requires multiple bits. The LRU and the Second Chance both require a relatively lengthy process of updating their method of determining the safety of replacing a page. Second Chance merely sets a single bit during one sweep of those pages. LRU requires the traversal of the entire list of pages, comparing each page to the current page believe to be the least recently used. Second Chance has the benefit that the first page found that is considered acceptable to paged out can be replaced. This costs less overhead.

1. Random

A program that does not exhibit any pattern in the pages it requests performs best with FIFO. This is because none of the three algorithms provide an improvement in paging performance for such a program. Therefore, it is best to choose the algorithm which costs less overhead, as any work done is ineffective and wasted.

***Part 3***

Page Faults for a 500 Length, 100 Page Virtual Address Space Program

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Replacement | Frames | Spatial Temporal | Spatial | Temporal | Random |
| Second Chance | 10 | 207 | 355 | 204 | 442 |
| Second Chance | 25 | 117 | 224 | 156 | 364 |
| Second Chance | 50 | 19 | 9 | 80 | 231 |
| Second Chance | 80 | 0 | 0 | 14 | 83 |
| FIFO | 10 | 130 | 118 | 214 | 446 |
| FIFO | 25 | 68 | 48 | 168 | 357 |
| FIFO | 50 | 13 | 3 | 91 | 227 |
| FIFO | 80 | 0 | 0 | 17 | 66 |
| LRU | 10 | 132 | 115 | 215 | 445 |
| LRU | 25 | 69 | 37 | 174 | 362 |
| LRU | 50 | 13 | 3 | 86 | 226 |
| LRU | 80 | 0 | 0 | 18 | 78 |

*Second Chance:*

The results for the Second Chance algorithm are not as good as I expected. I thought that, since Windows uses the Second Chance algorithm, it must be one of the best. However, in my simulation, it performed the worst when in three out of four conditions involving spatial or spatial-temporal locality. Perhaps Second Chance does not take advantage of the characteristics of spatial locality.

*FIFO:*

The results for FIFO were impressive. The overhead aspect of memory management was taken out of this experiment, which must be considered before concluding something from this data. However, according to the data, FIFO is as good as LRU on average.

*LRU:*

LRU did the worst when the program exhibited only temporal locality. This may just be due to random variation.

***Conclusion:***

Reference strings are a way to represent the unknown path of page requests a program will take along its execution. Attempting to artificially generate locality taught be about the characteristic patterns that program execution usually follows. It is not common for a program to execution similar to the random locality reference strings. Especially with modern compilers, code if highly optimized so that optimized replacement algorithms can optimize paging performance.

Considering the replacement algorithms assigned, I realize why operating system programming today in age is impossible to be done with less than a few hundred or thousand people. Every single decision involves a give and take.

Coding a demand paging memory management system was very fun. Then adding a round robin scheduler on top of that proved to be a very fun and stressful week. The most important thing I learned is the dependence that an Operating System has on memory. Due to the huge performance penalty, an Operating System must try to keep as much as possible in memory, but also be the most efficient about it as possible. Whatever memory the OS uses, it is memory it cannot use to increase performance of user programs.

Operating System code execution is unavoidable. However, by coding my own scheduling algorithm, I was able to catch an abstract glimpse of all the possible duties an OS must perform all the time in between user programs. I wish I had more time to continue building upon this project.

I have three planned next steps. The first is to code a small chance that a process will request a page outside its address space. My simulation would have to catch that and terminate the program. Another idea is to create other types of resources other than memory that reference strings can request. This would allow me to consider and code deadlock prevention and avoidance algorithms. Finally, I would have like to have implemented a page table. Due to time constraints, I implemented an inverse page table.